Accelerating Super-Resolution Reconstruction Using GPU by CUDA

Toygar Akgün and Murat Gevrekci

ASELSAN Microelectronics, Guidance and Electro-Optics Division,
Akyurt, Ankara / Turkey
{takgun,mgevrekci}@aselsan.com.tr

Abstract. This paper demonstrates a massively multi-threaded implementation of super-resolution image formation on the NVIDIA CUDA architecture. On the algorithm side maximum a-posteriori (MAP) reconstruction is adopted with sub-pixel translational motion estimation algorithm for spatial resolution enhancement. Resulting algorithm is implemented in CUDA using a low end $GT640$ GPU, and an overall speed up of $10 - 11$ times is achieved compared to ANSI C implementation running on a Core $i5$ CPU.

Keywords: Massive multi-threading, GPU, CUDA, super-resolution, multi-frame resolution enhancement.

1 Introduction

Graphic Processor Unit (GPU) had been used for general purpose programming since late 1990s by carefully leveraging OpenGL API. But neither OpenGL nor these older GPUs were designed with this goal in mind, leading to limited functionality and a steep learning curve. Starting early 2000s GPU evolved into a programmable, highly parallel, multi-threaded, many-core processor with tremendous computational float-point horsepower and very high memory bandwidth. In 2006 NVIDIA Corporation introduced the CUDA (Compute Unified Device Architecture) architecture with an accompanying programming model and API. CUDA and its accompanying API were *designed* to allow users with high computational needs to leverage GPU's compute power with minimal learning effort. Since then massive-multithreading on GPUs has been gaining traction and general purpose computing on GPUs is being utilized by researchers coming from vastly varying backgrounds [1].

This paper demonstrates a massively multi-threaded implementation of super-resolution reconstruction on the NVIDIA CUDA architecture. Super-resolution (SR) is a multi-frame video enhancement framework to obtain a high quality description from multiple degraded observations. SR reconstruction aims to compensate for image degradations i.e. aliasing, blurring, noise, interlacing, and low resolution. Sub-pixel shifts among consecutive frames are utilized to perform image enhancement. As discussed in [2], our multi-frame image enhancement system is composed of two main parts: registration and reconstruction. Work in

[2] is utilized for estimating the vertical and horizontal shifts among consecutive images. Reconstruction step uses a Bayesian framework to form a high-resolution image. This paper is structured as follows: We will first briefly go through the details of the super-resolution algorithm under discussion. Then we will discuss the CUDA mappings of the major processing blocks and present performance comparisons. Proposed methodology along with some visuals results are given in Section 2. Massively multi-threaded CUDA implementation of the algorithm is presented in Section 3 and performance boost of the proposed work is discussed in Section 4. The final section presents conclusions and future work.

2 Algorithm Details

Super-resolution system under discussion includes several building blocks. For a detailed discussion including references to prior art please refer to [2]. Consecutive images with various shifts are acquired using OpenCV image acquisition module. Pre-processing might be required to de-interlace in case interlacing is present. Registration step aims to align images on a common geometrical reference. Then a reconstruction step samples the aligned images on a sub-pixel grid to create finer details. Finally, contrast enhancement can be applied to emphasize local details as a post-processing step, which is planned as future work.

Representing the low-resolution degraded image as $\boldsymbol{y}_k$, and SR image (ground truth) as $\boldsymbol{x}$, image formation matrix can be represented as $\boldsymbol{H}_k$, which is the multiplication of down-sampling ($\boldsymbol{D}$), blurring ($\boldsymbol{B}_k$) and warping ($\boldsymbol{M}_k$) matrices.

$$\boldsymbol{y}_k = \boldsymbol{D}\boldsymbol{B}_k\boldsymbol{W}_k\boldsymbol{x} + \boldsymbol{n}_k = \boldsymbol{H}_k\boldsymbol{x} + \boldsymbol{n}_k, \tag{1}$$

$k = 1, ..., N$, where N is the number of images acquired and $\boldsymbol{n}_k$ is additive white Gaussian noise. Using Bayesian estimation, SR reconstruction can be written in the form of a cost function consisting of prior information and data fidelity terms. The optimization problem turns into following form using a discrete derivative operator ($\boldsymbol{L}$) as prior information

$$\boldsymbol{x}_{map} = \arg\min_{\boldsymbol{x}} \gamma^2 \parallel \boldsymbol{L}\boldsymbol{x} \parallel_2^2 + \sum_{k=1}^{N} \parallel \boldsymbol{y}_k - \boldsymbol{H}_k\boldsymbol{x} \parallel_2^2 . \tag{2}$$

Prior information provides smoothness to the SR estimate by penalizing high frequency components. Selecting the regularization parameter has critical importance to avoid over smoothing. During experiments we only kept data fidelity term in cost function, since analytic selection of regularization parameter (γ) is problematic and will violate the robustness we seek. The reduced cost function becomes:

$$C(\boldsymbol{x}) = \sum_i \parallel g_{r_i}(\boldsymbol{z}_i) - \boldsymbol{H}_i\boldsymbol{x} \parallel_2^2, \tag{3}$$

where $g_{r_i}(z)$ is the observation that passes through photometric conversion, also known as intensity mapping function, that compensates for the intensity fluctuations among infrared images. Time dependent intensity scaling or histogram equalization can be used for photometric mapping. This reduced cost function can be expanded using weighted least squares transform as

$$C(\boldsymbol{x}) = \frac{1}{2}\sum_i \left(g_{r_i}(\boldsymbol{z}_i) - \boldsymbol{H}_i\boldsymbol{x}\right)^T \boldsymbol{W}_i \left(g_{r_i}(\boldsymbol{z}_i) - \boldsymbol{H}_i\boldsymbol{x}\right). \tag{4}$$

Here $\boldsymbol{W}_i$ nothing but a certainty function of i^{th} image that weights the IR intensities to suppress dark noise. Cost function in Equation 4 represents a weighted least squares optimization. Taking certainty matrix $\boldsymbol{W}_i$ as identity turns the problem into regular least squares. Then the super-resolved output can be solved using iterative gradient descent techniques

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \gamma \sum_i \boldsymbol{H}_i^T \boldsymbol{W}_i \left(g_{r_i}(\boldsymbol{z}_i) - \boldsymbol{H}_i\boldsymbol{x}^{(k)}\right). \tag{5}$$

Here γ is the step size and taken as a constant in our experiments. SR algorithm is visualized in Figure 1 for sake of clarity. Forward projection matrix $\boldsymbol{H}_k$, is the building block of the algorithm as depicted in Figure 1. Forward projection is composed of down-sampling ($\boldsymbol{D}$), blurring ($\boldsymbol{B}_k$) and warping ($\boldsymbol{M}_k$) operations applied sequentially. Note that all of these operations are suitable for parallel implementation. Likewise, back projection $\boldsymbol{H}_k^t$ consists of up-sampling with zero insertion ($\boldsymbol{U}$), blurring ($\boldsymbol{B}_k^t$) and back warping ($\boldsymbol{M}_k^t$) operations. Blurring operation is same in case a symmetric kernel is adopted such as Gaussian.

Selected parameter set is given in Table 1 for future references. Parameters are kept constant throughout the experiments to demonstrate the robustness of the system.

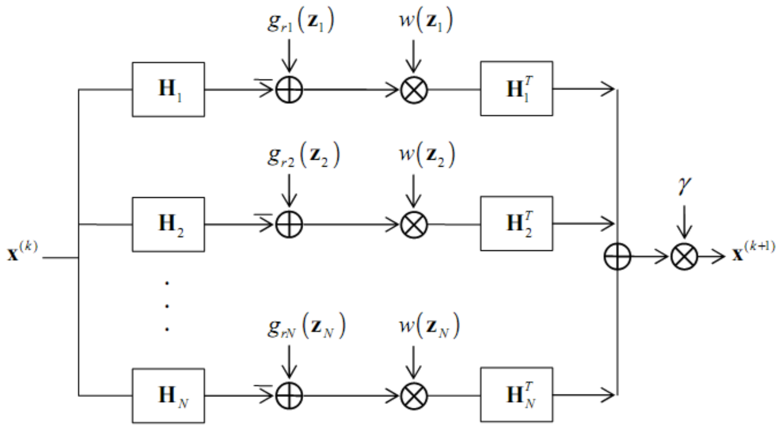

Fig. 1. SR system

Please note that vertical resolution enhancement factor should be selected twice as the horizontal resolution factor to compensate for the vertical decimation performed in preprocessing step in case frames are interlaced.

Table 1. Experiment setup

Number of low-res input frames	5
Vertical resolution enhancement factor	2
Horizontal resolution enhancement factor	2
Reconstruction iteration number	4
Gaussian kernel support	[5,5]
Gaussian sigma	1.0

Table 2. GPU specifications

CUDA cores	384
Graphics clock	900 MHz
Memory clock	900 MHz
Memory amount and type	2 GB DDR3
Memory interface width	128 bit
Memory bandwidth	28,5 GB/sec

Algorithm is suitable for parallel implementation as each frame and pixel inside the frames can be processed independently. In registration step translational shifts between input images and the reference image is computed independently. This independence paves the way for parallel computing in registration step. Reconstruction step is also parallel in nature since residual computation is independent for every input image. The scheme shown in Figure 1 illustrates how the reconstruction step is suitable for parallel implementation since we adopt an iterated back-projection method.

3 CUDA Implementation

The super-resolution algorithm described so far was implemented in CUDA to assess the potential performance boost. For the test results presented here a GT 640 NVIDIA GPU was used. The technical specifications of this card are as given in Table 2.

The competing platform is a Core i5 CPU clocked at 3,1 GHz with 8 GB RAM. Note that GT 640 is the smallest and weakest GPU from the Kepler architecture, which is the latest NVIDIA GPU architecture as of late 2012. As a result it has very low compute power and memory bandwidth compared to high end cards such as GTX 670, GTX 680 and GTX 690. In the following subsections we present the CUDA mappings of several major processing blocks. Note that there are several other CUDA kernels that will not be mentioned here, since they mostly handle simple data moving and updates.

3.1 Bilinear Scaling

Super-resolution reconstruction begins with an initial estimate image which is typically chosen as a bilinearly upscaled version of the original low resolution frame,

Table 3. Bilinear filtering code analysis

Thread numbers (x,y)	(32,32)
Shared memory bank conflict	N/A
Global memory BW efficiency	100%
Register usage	17
Occupancy	0,877
Total execution time per frame	161 microsec

where the vertical and horizontal scaling ratios are equal to the vertical and horizontal enhancement ratios. For the CUDA mapping of this bilinear upscaling block, the texture unit of the GPU hardware was used. Texture units are one of the available specialized hardware blocks on GPUs that handle simple pixel sampling operations. Such operations are quite common in graphics processing tasks. As a result, texture units are quite optimized and can provide substantial performance boosts. Extensive details of the texture hardware are beyond scope of this technical report, but we note one key property that was leveraged in this implementation. Texture units can bilinearly sample 2D arrays with very small computational load thanks to their specialized pixel sampling hardware. Initial implementation of the bilinear upscaling operation is presented below:

```
__global__ void bilinearResizeKernel(unsigned char *target,
                                     int width,
                                     int height,
                                     int pitch,
                                     float factor)
{
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x<width && y<height)
    {
        // Warped coordinates to sample from input
        float norm_x = ((float)x)/factor + 0.5f;
        float norm_y = ((float)y)/factor + 0.5f;

        // Read from texture and write to global memory
        target[y * pitch + x] =
                (unsigned char)(tex2D(texRef, norm_x, norm_y)*255.0f);
    }
}
```

As the code block presents, bilinear interpolation kernel is very simple. The heart of the kernel is the bilinear texture sampling operation (tex2D) at the very end. Since the output of the bilinear upscaling operation is also converted from 8 bit unsigned values to 32 bit float values for further processing, improvement is possible by slightly modifying the texture sampling operation to avoid an additional kernel launch. This faster implementation that combines bilinear interpolation with type conversion is presented in the code block below. Note that this very simple example demonstrates the difference between CPU and GPU programming in terms of surface access methodology.

```
__global__ void bilinearResizeToFloatKernel(unsigned char *target,
                                            float * target_f,
                                            int width,
                                            int height,
                                            int pitch,
                                            int pitch_f,
                                            float factor)
{
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x<width && y<height)
    {
        // warped coordinates to sample from input
        float normalized_x = ((float)x)/factor + 0.5f;
        float normalized_y = ((float)y)/factor + 0.5f;

        // Read from texture and write to global memory
        float temp = (tex2D(texRef, normalized_x, normalized_y)*255.0f);
        target_f[y * pitch_f + x] = temp;
        target[y * pitch + x] = (unsigned char)temp;
    }
}
```

Note that bilinear interpolation during texture fetch is only available for float type and there are some fine details regarding the setup and use of the texture units to get the best performance (including using surface writes to avoid additional CUDA surface copies). Thread mapping for the bilinear resampling kernel is very simple: Every thread handles a single output pixel. A slight speed-up (30 micro seconds faster) is possible by assigning four pixels to each thread, hence getting better memory bandwidth usage, but given texture units are cached and that bilinear sampling operation is called only once and constitutes only a small percentage of the overall execution time, the resulting speed-up would not be visible at the end. Execution configuration and performance figures for the second code block are given in Table 3.

Just to give an idea about how efficient this kernel is, the very same operation on a Core i5 CPU clocked at 3,1 GHz takes 5 ms, which means, for this specific block we have an average speed up of more than 30 times.

3.2 Image Warp

Image warp operation is modeled as a six parameter affine transform on the input image coordinates. Obviously, the transformed coordinates are not guaranteed to fall on a regular pixel grid. Off-grid pixel locations are simply obtained by bilinear interpolation that uses the four corner pixels of the grid block that includes the off-grid pixel location. As explained in the previous sub-section this is a perfect match for the texture hardware. The resulting CUDA implementation is presented in the code block below:

```
__global__ void warpAffineKernel4(float4 *target,
                                  float4 *grad,
                                  int width,
                                  int height,
                                  int pitch16)
{
   // Calculate normalized texture coordinates
   const int x = blockIdx.x * blockDim.x + threadIdx.x;
   const int y = blockIdx.y * blockDim.y + threadIdx.y;
   const int x4 = x * 4;

   float xf = (float)x4;
   float yf = (float)y;

   if(x<width && y<height)
   {
        float4 gradTemp = grad[y * pitch16 + x];
        // warped coordinates to sample from input
        float warped_x = c_H[0]*xf + c_H[1]*yf + c_H[2] + 0.5f;
        float warped_y = c_H[3]*xf + c_H[4]*yf + c_H[5] + 0.5f;
        float fx = tex2D(texRef, warped_x, warped_y);

        warped_x = c_H[0]*(xf+1.0f) + c_H[1]*(yf) + c_H[2] + 0.5f;
        warped_y = c_H[3]*(xf+1.0f) + c_H[4]*(yf) + c_H[5] + 0.5f;
        float fy = tex2D(texRef, warped_x, warped_y);

        warped_x = c_H[0]*(xf+2.0f) + c_H[1]*(yf) + c_H[2] + 0.5f;
        warped_y = c_H[3]*(xf+2.0f) + c_H[4]*(yf) + c_H[5] + 0.5f;
        float fz = tex2D(texRef, warped_x, warped_y);

        warped_x = c_H[0]*(xf+3.0f) + c_H[1]*(yf) + c_H[2] + 0.5f;
        warped_y = c_H[3]*(xf+3.0f) + c_H[4]*(yf) + c_H[5] + 0.5f;
        float fw = tex2D(texRef, warped_x, warped_y);

        // Read from texture and write to global memory
        target[y * pitch16 + x] = make_float4(fx,fy,fz,fw);
        grad[y * pitch16 + x] = make_float4(gradTemp.x-fx, gradTemp.y-fy,
                                            gradTemp.z-fz, gradTemp.w-fw);
    }
}
```

Note that this function is called at the inner most loop of the algorithm; hence it is required to be extremely optimized. As a result, the thread mapping is slightly modified compared to the bilinear interpolation kernel. Here each thread processes 4 pixels, which are modeled by the CUDA specific vector type float4.

Table 4. Affine warp code analysis

Thread numbers (x,y)	(32,32)
Shared memory bank conflict	N/A
Global memory BW efficiency	100%
Register usage	16
Occupancy	0,773
Total execution time per frame	245 microsec

To avoid an additional kernel launch, gradient surface update is also merged into this kernel by simply obtaining updated gradient values and conducting a global write. Execution configuration and performance figures for the previous code block are as given in Table 4.

3.3 Image Blur

Image blur block consists of 2D convolution with a separable kernel. The choice of a separable kernel is mostly due to its computational efficiency. For a 5×5 kernel, separable convolution requires 10 multiplications and 8 additions, whereas non-separable convolution would require 25 multiplications are 24 additions. Separable convolution is implemented as two consecutive kernel launches. The first kernel launch performs row-wise 1D convolution on the input image, and the second kernel launch performs column-wise 1D convolution on the output of the row-convolution kernel. Row and column convolution kernels are mostly adapted from NVIDIA's CUDA SDK with minor performance tunings. These kernels are presented in the following two code blocks.

As the code block below shows, the row convolution implementation uses shared memory. For the convolution operation shared memory usage provides substantial performance boost due to better memory bandwidth usage. To see this, simply consider what happens when we finish processing one pixel and move to the next pixel. The 5 pixel window that is used for filtering also shifts by one pixel and there is a 4 pixel overlap with the old filter window. Shared memory usage removes the need for dispatching a global read to obtain these pixels. At the beginning of the kernel each thread loads the main data to be processed by the thread block as well as the additional pixels that will be needed to obtain results at the boundary pixels. Note that row convolution operation uses a 5 tap blur filter and as a result boundary handling is required. This is simply due to the fact that the pixels at the thread block boundaries require 2 neighbors to their left and right sides. Finally, the filtering operation is performed and the results are written back to global memory. Note that every thread handles 4 pixels and loop unrolling (pragma unroll) is heavily used due to the fixed structures of the loops. Execution configuration and performance figures for row filtering are summarized in Table 5.

```
__global__ void convolutionRowsKernel(float *d_Dst,
                                      float *d_Src,
                                      int imageW,
                                      int imageH,
                                      int pitch)
{
    __shared__ float
    s_Data[BLOCKDIM_Y][(RESULT_STEPS + 2 * HALO_STEPS) * BLOCKDIM_X];

    //Offset to the left halo edge
    const int baseX = (blockIdx.x * RESULT_STEPS - HALO_STEPS) * BLOCKDIM_X + threadIdx.x;

    const int baseY = blockIdx.y * BLOCKDIM_Y + threadIdx.y;

    d_Src += baseY * pitch + baseX;
    d_Dst += baseY * pitch + baseX;

    //Load main data
    #pragma unroll
    for(int i = HALO_STEPS; i < HALO_STEPS + RESULT_STEPS; i++)
        s_Data[threadIdx.y][threadIdx.x + i * BLOCKDIM_X] = d_Src[i * BLOCKDIM_X];

    //Load left halo
    #pragma unroll
    for(int i = 0; i < ROWS_HALO_STEPS; i++)
        s_Data[threadIdx.y][threadIdx.x + i * BLOCKDIM_X] =
        (baseX >= -i * BLOCKDIM_X ) ? d_Src[i * BLOCKDIM_X] : 0;

    //Load right halo
    #pragma unroll
    for(int i = HALO_STEPS + RESULT_STEPS; i < HALO_STEPS + RESULT_STEPS + HALO_STEPS; i++)
        s_Data[threadIdx.y][threadIdx.x + i * BLOCKDIM_X] =
        (imageW - baseX > i * BLOCKDIM_X) ? d_Src[i * BLOCKDIM_X] : 0;

    //Compute and store results
    __syncthreads();
    #pragma unroll
    for(int i = HALO_STEPS; i < HALO_STEPS + RESULT_STEPS; i++){
        float sum = 0;

        #pragma unroll
        for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
            sum += c_Kernel[KERNEL_RADIUS - j]
                          * s_Data[threadIdx.y][threadIdx.x + i * BLOCKDIM_X + j];

        d_Dst[i * BLOCKDIM_X] = sum;
    }
}
```

As the following code block shows, the column convolution implementation uses shared memory, due the reasons discussed previously. At the beginning of the kernel each thread loads the main data to be processed by the thread block as well as the additional pixels that will be needed to obtain results at the boundary pixels. Note that just like row convolution, column convolution operation uses a 5 tap blur filter and as a result boundary handling is required. This is simply due to the fact that the pixels at the thread block boundaries require 2 neighbors to their top and bottom. Finally, the filtering operation is performed and the results are written back to global memory. Note that every thread handles 8 pixels and loop unrolling pragma unroll is heavily used due to the fixed structures of the loops. Execution configuration and performance figures column filtering are given in Table 6.

```
__global__ void convolutionColumnsKernel(float *d_Dst,
                                         float *d_Src,
                                         int imageW,
                                         int imageH,
                                         int pitch)
{
    __shared__ float s_Data[BLOCKDIM_X][(RESULT_STEPS + 2 * HALO_STEPS) * BLOCKDIM_Y + 1];

    //Offset to the upper halo edge
    const int baseX = blockIdx.x * BLOCKDIM_X + threadIdx.x;
    const int baseY = (blockIdx.y * RESULT_STEPS - HALO_STEPS) * BLOCKDIM_Y + threadIdx.y;

    d_Src += baseY * pitch + baseX;
    d_Dst += baseY * pitch + baseX;

    //Main data
    #pragma unroll
    for(int i = HALO_STEPS; i < HALO_STEPS + RESULT_STEPS; i++)
        s_Data[threadIdx.x][threadIdx.y + i * BLOCKDIM_Y] = d_Src[i * BLOCKDIM_Y * pitch];

    //Upper halo
    #pragma unroll
    for(int i = 0; i < HALO_STEPS; i++)
        s_Data[threadIdx.x][threadIdx.y + i * BLOCKDIM_Y] =
        (baseY >= -i * BLOCKDIM_Y) ? d_Src[i * BLOCKDIM_Y * pitch] : 0;

    //Lower halo
    #pragma unroll
    for(int i = HALO_STEPS + RESULT_STEPS; i < HALO_STEPS + RESULT_STEPS + HALO_STEPS; i++)
        s_Data[threadIdx.x][threadIdx.y + i * BLOCKDIM_Y] =
        (imageH - baseY > i * BLOCKDIM_Y) ? d_Src[i * BLOCKDIM_Y * pitch] : 0;

    //Compute and store results
    __syncthreads();
    #pragma unroll
    for(int i = HALO_STEPS; i < HALO_STEPS + RESULT_STEPS; i++){
        float sum = 0;
        #pragma unroll
        for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
            sum += c_Kernel[KERNEL_RADIUS - j]
                        * s_Data[threadIdx.x][threadIdx.y + i * BLOCKDIM_Y + j];

        d_Dst[i * BLOCKDIM_Y * pitch] = sum;
    }
}
```

Table 5. Row filtering code analysis

Thread numbers (x,y)	(32,4)
Shared memory bank conflict	0%
Global memory BW efficiency	100%
Register usage	22
Occupancy	0,957
Total execution time per frame	120 microsec

Table 6. Column filtering code analysis

Thread numbers (x,y)	(32,8)
Shared memory bank conflict	0%
Global memory BW efficiency	100%
Register usage	32
Occupancy	0,476
Total execution time per frame	140 microsec

4 Performance Boost and Comparison

In this section we present the overall performance boost obtained by the CUDA implementation. The results we compare against are obtained by a Core i5 processor clocked at 3,1 GHz with 8 GB RAM for images of dimension 384x256 (single channel, 8-bit).

– One iteration of the gradient computation on the CPU was timed to be around 18 ms.
– One iteration of the gradient computation on the GPU was timed to be around 2 ms.

Gradient computation is the main computational block of the super-resolution algorithm and consists of applying the forward imaging model, taking the difference between the resulting *constructed* observation and the corresponding true observation, and finally applying the backward imaging model on the error. Hence, any speed up obtained for this block has a direct effect on the overall execution time. Please note that the "QueryPerformance" functions provided by MS has a time resolution of 1 ms. We have reason to believe that actual GPU timing is somewhere between 1ms and 2 ms, but we will not discuss this here and simply assume a 9X speed up.

– The overall algorithm execution time per frame on CPU is between 410 - 440 ms.
– The overall algorithm execution time per frame on GPU is between 42 - 46 ms.

These numbers represent a worst case speed up factor of approximately 9X and a best case speed up factor of approximately 10X on a low end GT 640 GPU, which has about 3 times lower core clock rate and 4 times less memory compared to the competing CPU configuration. For GT 640, current performance profiling results show that the CUDA implementation is both memory bandwidth and compute limited. This suggests that a high end GPU is guaranteed to improve the overall performance. Just to give an idea, GTX 670 has 3,5 times more compute power and 6,5 times higher memory bandwidth compated to GT 640.

5 Conclusion and Future Work

In this paper we discussed a massively-multithreaded CUDA implementation of the super-resolution algorithm originally presented in [2]. Performance test results were presented comparing a low-end $GT640$ NVIDIA GPU to a Core i5 CPU. Current implementation does not support overlapping kernel execution with data read/write. As future work, the current implementation will be moved to a higher end GPU such as $GTX660Ti$, $GTX670$ or $GTX680$ and modified to overlap kernel executions with data copies. We expect these modifications to further enhance the overall performance and lower the total execution time per frame to below 10 ms per frame, allowing the final implementation to run at 60 frames per second for frames of size 384x256 with a resolution enhancement factor of 2X.

References

1. NVIDIA CUDA C Programming Guide, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
2. Gevrekci, M., Gunturk, B.K.: Image Acquisition Modeling for Super-Resolution Reconstruction. In: IEEE Int. Conf. on Image Processing (ICIP), vol. 2, pp. 1058–1061 (September 2005)